GPU COMPUTING FOR PARTICLE TRACKING [*]

Hiroshi Nishimura [†], Kai Song, Krishna Muriki, Changchun Sun, Susan James, Yong Qin
Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720, USA

Abstract

This is a feasibility study of using a modern Graphics Processing Unit (GPU) to parallelize the accelerator particle tracking code. To demonstrate the massive parallelization features provided by GPU computing, a simplified TracyGPU program is developed for dynamic aperture calculation. Performances, issues, and challenges from introducing GPU are also discussed.

INTRODUCTION

General-purpose Computation on Graphics Processing Units (GPGPU) bring massive parallel computing capabilities to numerical calculation [1]. However, the unique architecture of GPU requires a comprehensive understanding of the hardware and programming model to be able to well optimize existing applications. In the field of accelerator physics, the dynamic aperture calculation of a storage ring, which is often the most time consuming part of the accelerator modeling and simulation, can benefit from GPU due to its embarrassingly parallel feature, which fits well with the GPU programming model. In this paper, we use the Tesla C2050 GPU which consists of 14 multi-processors (MP) with 32 cores on each MP, therefore a total of 448 cores, to host thousands of threads dynamically. Thread is a logical execution unit of the program on GPU. In the GPU programming model, threads are grouped into a collection of blocks. Within each block, multiple threads share the same code, and up to 48 KB of shared memory. Multiple thread blocks form a grid, which is executed as a GPU kernel. A simplified code that is a subset of Tracy++ [2] is developed to demonstrate the possibility of using GPU to speed up the dynamic aperture calculation by having each thread track a particle.

PARTICLE TRACKING

Accelerator Code

A new library TracyGPU is developed to demonstrate the above concept. Among several programming models that are available to GPU, Compute Unified Device Architecture (CUDA) [1] is chosen to create a subset of Tracy++, which is a C++ code developed and used by Advanced Light Source (ALS) [3]. The original ALS storage ring lattice [3] is used for testing. This work is developed with CUDA C, which is a variant of C that supports both CPU and GPU. Theoretically, it is possible to port the entire C++ library including data structure and classes to GPU. However, as we describe in this paper, due to the limitations of the CUDA compiler and the GPU architecture, TracyGPU library is a redesigned version of Tracy++ to utilize both the CPU and GPU for accessibility, simplicity, and flexibility.

TracyGPU contains three logic units: Optics, Fitting, and Tracking. The Optics unit calculates the accelerator optics functions such as beta and dispersion functions, betatron tunes, chromaticities and the natural emittance, by using the matrix formalism. The Fitting unit fits optics functions and properties by calling Optics unit. Tracking unit tracks particles to calculate dynamic apertures.

TracyGPU is developed with two execution modes for performance comparison: 1) all three logic units run on CPU, 2) Optics and Fitting run on CPU while Tracking runs on GPU. In the second mode, due to some physical limitations that are discussed later in this paper, only the Tracking unit is currently ported to GPU, while the other two units remain on CPU and are not parallelized. To better utilize the limited shared memory on GPU, another data structure uRing, which is different than the original Ring data structure that is dedicated to CPU, is designed for GPU. The new uRing data structure only has a 7 KB of memory footprint. By comparing it to the 42 KB memory footprint of Ring, it stores more efficiently and saves memory spaces for other data structures.

Steps of the Calculation

When calculating the dynamic apertures for a given optics setting, particles are tracked at 651 transverse mesh points (x, y) with x=0, 1, 2, .. , 30 mm and y=0, 1, 2, .. , 20 mm. This calculation is performed in parallel on GPU by creating 651 threads. Each thread tracks a particle with a given initial point that is up to 400 turns by checking the transverse excursion. These 651 threads form a thread block.

To further explore the GPU parallelization capability, we launch multiple Ring settings to calculate the dynamic apertures of each Ring, *e.g.*, 100 different sets of betatron tunes. If a GPU can host multiple optics settings simultaneously, the entire tracking process can be done in parallel.

The actual calculation works as described below. A Ring object is created on CPU. The optics are calculated and fitted by using the Optics and Fitting units. This is done for all the optics settings, and the output magnet settings are stored in a data array. This data array and a subset of Ring data structure, uRing, will be passed to GPU. On GPU, each thread block saves a copy of the uRing in its shared memory, and the magnet values will be set from the input

[*] Work supported by U.S. DoE Contract No. DE-AC02-05CH11231
[†] H_Nishimura@lbl.gov

data array. This results in each thread block having its own uRing object settings, thus being able to calculate dynamic aperture independently and in parallel.

RESULTS AND DISCUSSION

Performance Measurement

As mentioned previously, TracyGPU library has two execution modes. The wall clock time spent on the TracyGPU code is compared between these two execution modes to demonstrate the performance improvement. Figure 1 shows that the total simulation time scales as N increases for both mode 1 and mode 2, in which N represents the number of Rings. It is observed that running on GPU takes about one order of magnitude less time than solely on CPU, even though only the Tracking unit is ported to GPU.

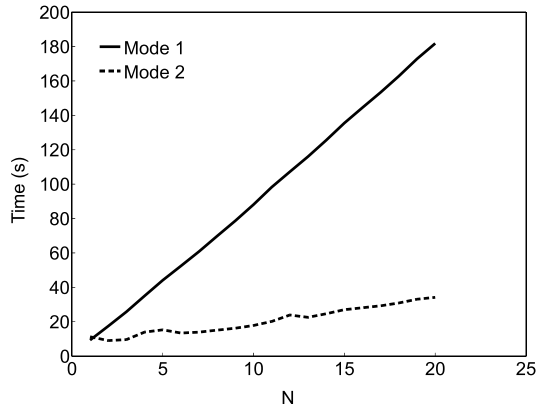

Figure 1: Execution time comparison between the CPU and GPU versions of TracyGPU

Figure 2 shows the run time further broken down for different units in execution mode 2, in which the solid line depicts the Optics and Fitting units running on CPU, and dashed line represents the Tracking unit running on GPU. As indicated, the time spent on the Tracking unit increases little as the number of Rings is increased. This is due to the massive parallelization and the large number of cores available on the Tesla C2050 GPU. However, for the Optics and Fitting units, since they are only executed on CPU and these portions of code are not in parallel, the run time increases at a larger O(N) scale. As a consequence, this will affect the overall performance when a large N value is used.

To further analyze the performance profile on GPU, we also separately measured the kernel execution time and the time spent on initializing the CUDA environment. Because CUDA API does not provide an explicit way to initialize its environment, the latter part is roughly measured by using the time spent on the first cudaMalloc() function, which is usually the first CUDA API called in the program. As showed in Fig. 3, CUDA initialization time typically takes

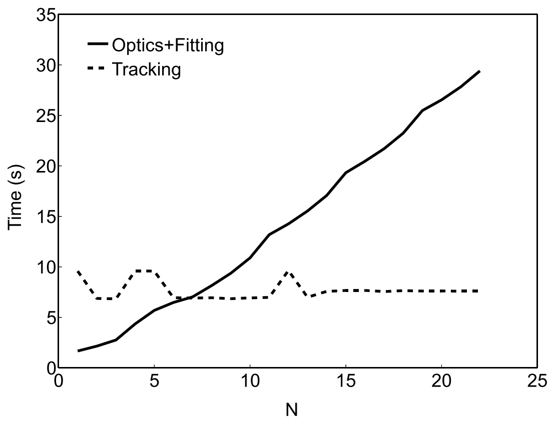

Figure 2: Execution time of Optics/Fitting and Tracking units for TracyGPU running at GPU mode

about 6 seconds and could randomly go up to about 8.5 seconds. The reason for the randomness is unknown. However it is suspected that this is related to the fact that we have two C2050 on the host and it may not be initializing the same device each time. Nevertheless, this initialization time also reduces the overall performance, especially when less Rings are used. As also indicated in Fig. 3, the time spent in the GPU kernel routines does not increase linearly but exhibits a step increase with a period of 14 Rings. This is due to the 14 multi-processors available on the GPU. During each iteration, a total of 14 Rings are launched simultaneously on these 14 multi-processors to run in parallel, and synchronized at the end of each iteration. Thus they consume the same amount of time to finish.

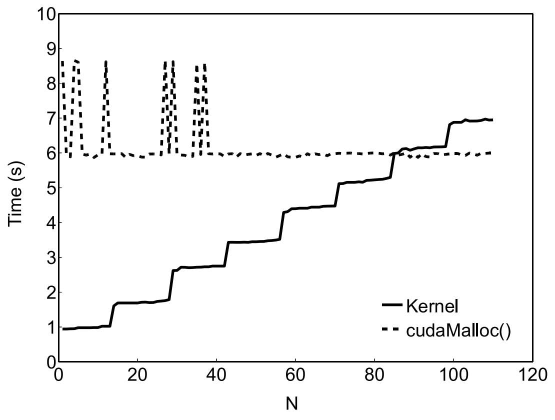

Figure 3: GPU initialization and kernel time comparison

With all these unique performance characteristics, it is also worthwhile to assess the overall speedup versus the speedup only for the Tracking unit, which is the only part that is ported to GPU. In Fig. 4, overall speedup is defined as the run time of execution mode 1 divided by run time of execution mode 2 which are showed in Fig. 1; while kernel speedup is defined as the execution time of the Tracking unit running on CPU divided by the execution time running on GPU. The results indicate that the overall speedup is considerably less than that of the GPU kernel due to the non-optimized Optics and Fitting units, as well as the GPU initialization time.

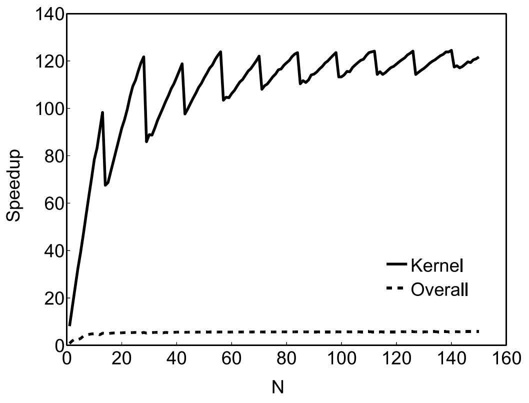

Figure 4: TracyGPU speedup comparison

Issues and Lesson Learned

There are two major issues when porting Tracy++ library to GPU: 1) dealing with data structures with many pointers and classes and, 2) memory limitation of the GPU kernel. For these reasons, the whole Ring data structure was rewritten to eliminate classes, and to pre-allocate all data arrays inside the Ring. Another obstacle that is noticed during the development is that, it is not possible to compile the code if we try to fit the entire library into the kernel. This is because some of the library functions are too big to optimize, thus the compilation will eventually run out of memory and fail. As a result, we removed Optics and Fitting units from the GPU kernel and performed these calculations on CPU to further shrink the GPU kernel size.

With all these challenges, it is learned that porting existing C/C++ library to GPU is not trivial. It usually requires partial or full reconstruction of the data structures and functions so that the kernel can be successfully compiled and efficiently executed on GPU. Therefore, someone with a comprehensive insight of the library can be extremely helpful. Based on our experience, existing computation libraries, especially those designed without having GPU architecture in consideration, could be difficult to port due to the differences between CPU and GPU architectures. One solution for this is to start by re-evaluating the design of the original data structures and algorithm, and optimize specifically for GPU.

Future Work

In this study, a simple GPU based particle tracking code, TracyGPU, is developed, and the performance characteristics are studied. The same technology could also be extended to other potential applications, *e.g.*, particle tracking for Frequency Map Analysis (FMA) [4, 5]. By adding two extra arrays to the kernel that performs the particle tracking, the particle trajectories can be maintained and then transferred from GPU to CPU for FMA. The code demonstrated here is implemented with single-precision floating-point calculations. However we are also aiming at a double-precision version to better suit accelerator simulations. In this paper we presented a simplified test case with up to several hundred Rings. As discussed above, when more and more Rings are added to the simulation, which will be the typical usage of TracyGPU, Optics and Fitting units will consume a significant portion of the total simulation time. Nevertheless, since these two units are currently running on CPU and in serial, parallelization technologies for CPU, such as OpenMP, multi-threading, or vectorization could also be used for further optimization and performance improvements.

SUMMARY

In this paper we performed the feasibility study of porting an existing Tracy++ library to Nvidia Tesla C2050 GPU for ALS lattice research. Performance of the new TracyGPU library, as well as the challenges encountered during this effort are extensively discussed. Based on this preliminary research, good performance enhancement is observed, however further optimization and performance improvements are also expected. One potential application of this library is provided as the next step research direction. From this study, we anticipate that the massive parallelization features introduced by GPGPU could greatly benefit the HPC community and facilitate scientific research.

ACKNOWLEDGEMENTS

This research used resources provided by LRC, Laboratory Research Computing [6], which is managed by LBNL IT division. The authors would also like to thank David Robin from ALS and Gary Jung from IT Division for their encouragement and support, as well as John Stratton from UIUC for his advice on GPU programming.

REFERENCES

[1] David B. Kirk, Wen-mei W. Hwu, “Programming Massively Parallel Processors, ISBN: 978-0-12-381472-2, 2010

[2] H. Nishimura, PAC 2001, Chicago, June 2001.

[3] LBL PUB-5172 Rev. LBL,1986. A. Jackson, IEEE 93PAC, 93CH3279-7,1432, 1993.

[4] J. Laskar, Icarus **88**, 266-291, 1990.

[5] D. Robin *et al.*, Phys. Rev. Lett. **85** (2000) 558.

[6] Laboratory Research Computing, `http://lrc.lbl.gov`